Master

Combining Local Negotiation And Global Planning In Cooperative Software Development Projects

Kazuo Okamura

CCS TR #163, Sloan School WP #3662-94
August 1993

CENTER FOR
COORDINATION
SCIENCE

Massachusetts Institute of Technology
Sloan School of Management
Cambridge, Massachusetts

Combining Local Negotiation And Global Planning In Cooperative Software Development Projects

Kazuo Okamura

CCS TR #163, Sloan School WP #3662-94
August 1993

Combining Local Negotiation And Global Planning In Cooperative Software Development Projects

Kazuo Okamura
Center for Coordination Science
Massachusetts Institute of Technology
One Amherst Street (E40-179)
Cambridge MA 02139

Information Systems Research Laboratory
Matsushita Electric Industrial Co., Ltd.
1006 Kadoma, Osaka 571 JAPAN
Phone: +81-6-906-2443 Email: okamura@isl.mei.co.jp

August 1993

ACKNOWLEDGMENT

The author would like to thank Thomas Malone for his generous advice. Bob Halperin provided helpful comments on earlier drafts of this paper. The research support of the Center for Coordination Science at the Massachusetts Institute of Technology is gratefully acknowledged.

Combining Local Negotiation And Global Planning In Cooperative Software Development Projects

ABSTRACT

In cooperative software development, each programmer has their own plans and conflicts or redundancies inevitably arise among them. We are concerned with two main problems: first, to control changes without sacrificing programmers' flexibility, and, second, to guide change activities to conform project policies. Traditional methods of change request management focus on the management process structure based on project policies while cooperative development methodologies concern mainly with the conflict resolutions among each changes. In this paper, we describe an architecture which deals with proposal of changes. Based on plan integration it seamlessly supports both change coordination through negotiations and the change management process to have changes converge until they meet the project goals.

To appear in Proceedings of the COOCS 93 ACM Conference on Organizational Computing Systems
Milpitas (near San Jose) California, November 1-4, 1993

INTRODUCTION

One of the major problems in software projects is how to manage changes to software artifacts. Programmers change software based on certain plans. In cooperative software development, in which parallel development is allowed as much as possible, each programmer may take actions based on their own plans. Sooner or later, conflicts or redundancies will inevitably arise among these plans. Some of them might compete with each other for software modules while others might have the common goals. These problems hinder the programmers' activities and reduce the project's performance and productivity. A proper change control mechanism is, therefore, a crucial part of cooperative software development environments.

In this paper, we are concerned with the two main problems in change control:

1. To control changes without sacrificing programmers' flexibility
2. To guide programmers' change activities to follow project policies

Studies of cooperative software development mainly discuss the first problem. Several systems [22, 1] help programmers merge their work when conflicts are found. Kaiser [12] extends the traditional database transaction model to give programmers flexible controls over transactions. Softbench [4] provides an event-based architecture in which programmers are notified when some state changes, such as changes to source code, or tool invocations. In these systems, it is the programmers that are responsible for resolving conflicts among their tasks. In the traditional configuration management methodology [2, 25], the second problem is treated as configuration control which focuses on the change request management process. The elements and the structure of the process are rigidly defined so that management tools can assist change request managers or the configuration control board in controlling programmers' change activities to comply with the project constraints. If, however, change request management is performed only from the project viewpoint, it can easily becomes a bureaucratic system [11]. Therefore it is very important to consider programmers' individual viewpoints and try to avoid diverting the programmers from their creative activities.

With the pros and cons of these methodologies in mind, our approach to the problems is an attempt to extend change request management concepts based on plan integration framework [24] in the AI planning paradigm. In this paper, we wish to describe an architecture which deals with proposal of changes. It seamlessly supports both change coordination by programmers and the management process to have changes converge until they meet the project goals.

We first describe the concepts underlying our approach and outline our assumptions about the context of the architecture. Then we describe the PCRM(Plan-based Change Request Management) system which is an experimental implementation of our architecture and discuss its three aspects: the change plan

integration algorithm, conflict and redundancy resolution through negotiations, and customizable representation of a change management process.

CHANGE REQUEST MANAGEMENT BASED ON PLAN INTEGRATION

A typical change request form [25] in traditional configuration management is essentially an informal plan description for the change because it contains the information such as the purpose of the change, items to be changed, when the change can be done, and the expected effect of the change. These correspond to a goal, resources, a pre-condition and a post-condition of a plan description, respectively. Therefore, change request management can be performed through integrating change plans into a consistent project plan for the changes.

Plan integration is usually useful in a relatively static environment. Software development is not such a environment because it is difficult to discern and/or articulate all the constraints a-priori for a software system [21]. However, when the assumptions made at a particular integration point is invalidated by later changes in the environment, one can express the discrepancies as new change requests, which will be inputs to the next plan integration. Therefore, if the integration procedure could be done easily and repeatedly, plan integration would be a worthwhile support for change control in software development. Not only the result of the integration but also the process itself would be very useful. If it could be done through social protocols, programmers could reach an agreement about change activities in the project.

In our architecture, change plans are written by programmers who concentrate on their own goals and are then passed to the change request management system that performs the integration. The plan integration process proceeds with checking relationships among plans to find conflicts and redundancies, and resolves them. Programmers can take part in the resolution process, and, therefore, the management system can respect their original intentions as much as possible. It is also important for the management system to resolve the conflicts and the redundancies in such a way that the change plans can comply with project policies. The integration process with such features ensures that every affected programmer has reached global consensus on the integrated plan for changes which meets the project's goals.

Change plan integration in the PCRM system has the following features:

(1) Selection strategies of actions to resolve conflicts and redundancies. The system has flexible strategies to select a list of possible actions to resolve conflicts and redundancies.

(2) Resolving conflicts and redundancies through negotiations. The system mediates between the programmers by creating a communication channel and suggesting the possible actions.

(3) A tailorable representation of project's change management policies. Several levels of customization are provided to tailor the system according to project requirements.

Our Assumptions

We make several assumptions which allow us to concentrate on the problems in change request management.

- Some form of version and configuration management system (such as RCS, CVS [3]) is available to identify software items to be changed, and it can be accessed by the PCRM system.

- Programmers are in a networked environment which allows them to work in both their local work space as well as the global space provided by the version and configuration system. Whenever they want to make changes to software items in the global space, they have to propose the changes.

- There exists a base system (which, for example might be the previously released system) from which the new system is evolved.

- One of the programmers operates the PCRM system as a Change Request manager (the CR manager).

OVERVIEW OF THE PCRM SYSTEM

Figure 1 shows how change request information is processed in the PCRM system.

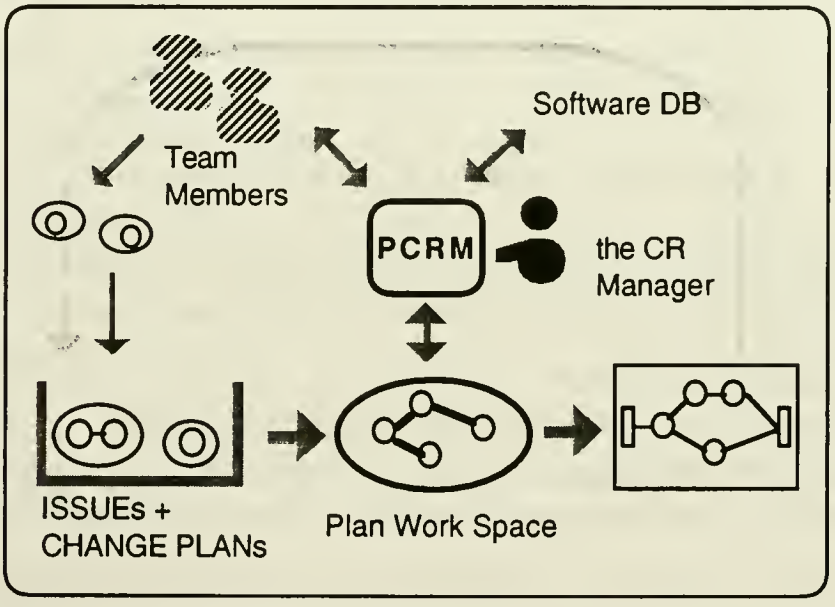

Figure 1: Change Request Management with the PCRM System

1. Initiation of Change Plans

A change request process starts when programmers in a project receive problem reports from users of a target software or write their own reports. Then the programmers screen these informal reports and turn them into two formal descriptions called ISSUEs and CHANGE PLANs. An ISSUE is a problem description that includes the information about a type of the problem, software items related to the problem, the original reporter, and the originator of the ISSUE. A type of a ISSUE can be either a bug, an

enhancement or a refinement. A CHANGE PLAN is a plan description which consists of CHANGE TASKs and a goal that is to solve an ISSUE. For example, the goal could be to fix a bug or enhance some functionality. One problem report might be turned into several ISSUEs and solving one ISSUE might require several CHANGE TASKs in a CHANGE PLAN. Conversely, a single CHANGE TASK would solve several ISSUEs. One of the simplest examples is a CHANGE PLAN with a single CHANGE TASK which solves an ISSUE of a software bug. A complex example is a CHANGE PLAN for a function enhancement in the next software release, with a set of CHANGE TASKs of several programmers.

2. Preparation for the Integration

The CR manager in the project collects CHANGE PLANs from programmers and screening ISSUEs to identify relationships among them. The most important relationship is the equality among them. The CR manager can reject ISSUEs and the related CHANGE PLANs at this point. A set of the CHANGE PLANs and valid ISSUEs are the results of the preparation.

3. Integration of Change Plans

The PCRM system starts integration of the given CHANGE TASKs. Whenever a conflict or a redundancy is found, the system creates a list of possible actions based on project policies and executes one of the actions which is determined by negotiations among the programmers. After resolving every conflict and redundancy, the system outputs an integrated plan for the requested changes. This plan may be an input to another integration if necessary.

The Architecture of the PCRM System

The architecture of the PCRM system is shown in Figure 2. The main part of the system consists of a set of agents: the plan integrator, the negotiator, the knowledge base, and the user interface agent. The kernel of the system is the plan integrator which contains the consistency checker, the conflict resolver, and the action executor. We have enhanced Oval [16] with a Prolog-based inference mechanism and implemented these agents as special Oval agents. Users of the system can create change-related descriptions as semi-structured Oval objects and exchange them over a network using Oval's communication mechanism, which has a conversation tracking function similar to the Coordinator [27].

In the rest of the paper, we will describe in details how the PCRM system integrates CHANGE PLANs into a consistent plan.

INTEGRATION OF CHANGE PLANS THROUGH NEGOTIATIONS

The basic plan integration algorithm used by the PCRM system is an enhanced version of [24]. His framework includes a mechanism for reasoning about resources focusing on situation-dependent operators (i.e. CHANGE TASKs in our case) which consume and produce resources. Therefore it provides a good theoretical basis for our approach in which handling software resources is an indispensable aspect of CHANGE TASKs. There are three major enhancements in our algorithm:

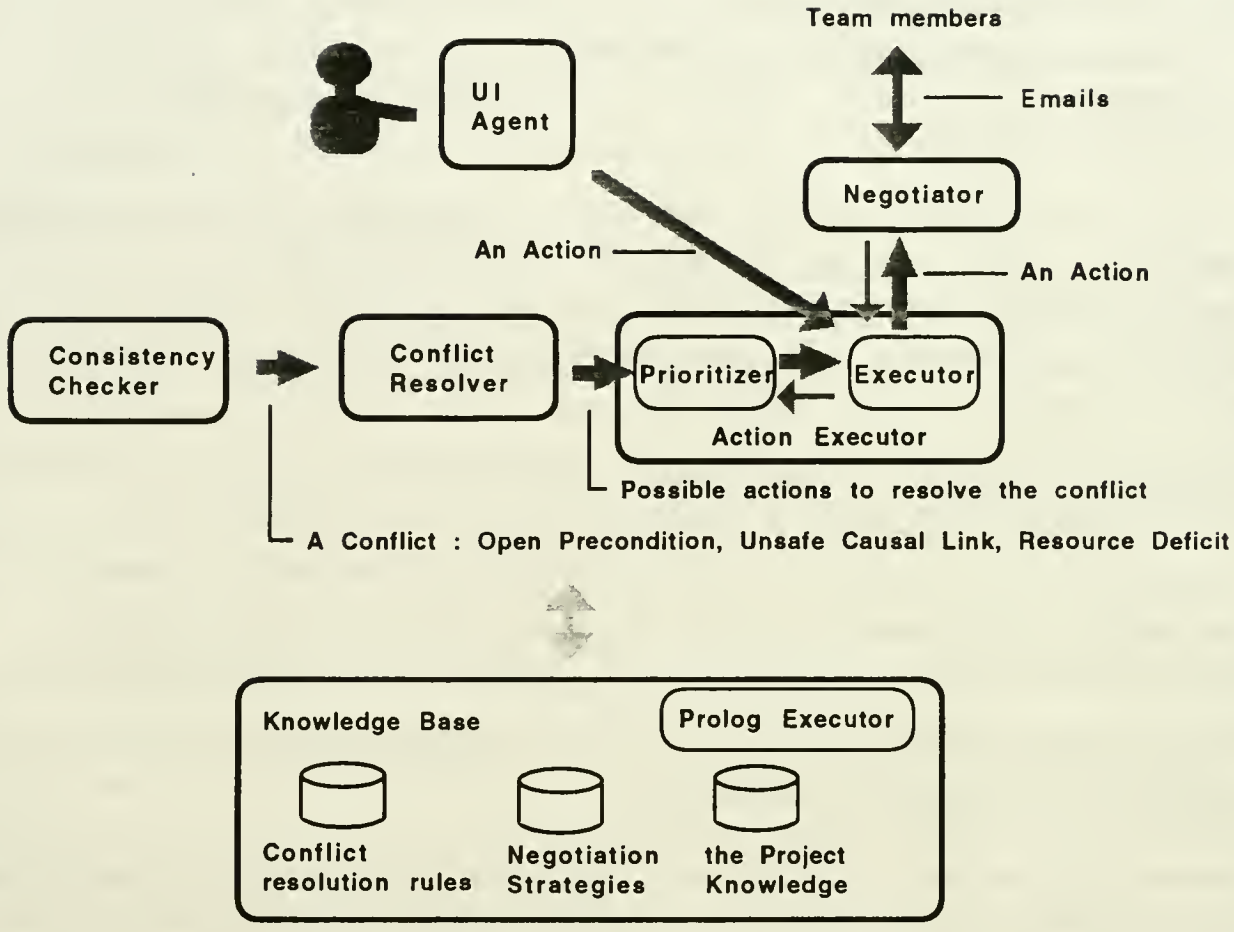

Figure 2: The PCRM System Architecture

1) Conflict resolution strategies based on domain knowledge. We enhanced the original algorithm by adding tailorable strategies to include project policies for managing change requests.

2) Conflict and redundancy resolution through negotiations. The original algorithm constructs integrated plans by applying every possible conflict resolution operations without user interaction. In change request management, however, it is highly desirable to have affected programmers participate in the decision making process so that they can arrive at a consensus on how to resolve conflicts. We have enhanced the algorithm by adding a negotiation support mechanism for that purpose.

3) Handling software resources. In the original algorithm, a resource has its amount and is consumable. However, if a resource is, for example, a software module, one cannot consume it, nor change the amount of it, in the original sense. We have extended definitions of consumption and amounts of resources.

Although discussing correctness of the algorithm is beyond the scope of this paper, the enhancements do not affect the correctness of the original algorithm since our algorithm is a domain-specific variant of the original one.

REPRESENTING CHANGE PLANS

In this section we formally define CHANGE PLANs.

A CHANGE PLAN consists of:

1) A set of CHANGE TASKs or tasks. Each task must have a goal, preconditions, and effects. There are two special tasks: the initial task I and the final task G[1]. I has no precondition, and has the effect of asserting the propositions and producing resources in the amounts specified in the initial state. G has no effects, and has preconditions specifying the propositions and lower bounds on resource production specified in the goal state.

2) A partial order < on tasks. If i < j then i must precede j in any total ordering that extends the partial order. The initial task I must precede every other task, and the final task G must follow every other task. We say that each producer P of some resource r is a supplier for each consumer C of r that follows P, and each consumer C of some resource r is a competitor for any consumer C' of r that might follow C. Two consumers of the same resource that are unordered with respect to each other are mutual competitors.

3) A set of causal links of the form <i, p, j> where i and j are tasks such that i < j, i asserts p, and p is a precondition of j. We say that i is the establisher of j.

4) Associated with each task S is the guaranteed resource supply (GRS), a lower bound on the level of each resource in the input state of S (the state immediately preceding S's execution). The GRS value of S with respect to resource r can be calculated as the amount of r produced by the suppliers of r that must precede S minus the amount of r consumed by the competitors of r that might precede S.

5) A software resource is modeled as a resource that is consumed and produced in equal amounts by a CHANGE TASK which uses it. The CHANGE TASK may also produce a new version of the resource .

6) Amount of a software resource is used to indicate the number of parallel tasks, that are allowed to be simultaneously executed using the resource. The initial amount is always 1. For example, 2 CHANGE TASKs A and B can be executed at the same time using a module M if the amount of M is increased to 2 and the GRS values in A and B are both 1. Although how to execute these parallel tasks with the same resource depends on execution environments, A and B may create a binary branch in a version tree of the module M.

---

[1] To make a CHANGE PLAN description simpler, the PCRM system allows user to write a CHANGE PLAN without specifying the initial and final tasks. In such a case, the system automatically adds the two special tasks to the CHANGE PLAN at the integration time.

7) The amount of a software resource cannot be increased more than the number of tasks allowed to be executed at the same time. If such a task is withdrew after the amount of the resource is increased, the amount should be decreased. The amount of a software resource is always related to the number of the existing tasks which use the resource and there should be no "stock" of it.

CONFLICTS AND REDUNDANCY

Our algorithm first initializes a plan by creating two special CHANGE TASKs: I and G. I is the initial producer of all relevant resources and has the effect of asserting all ISSUEs. ISSUEs are consistent at this point because they have been already screened by the CR manager. G has a precondition which includes the aggregate goals of CHANGE PLANs to be integrated. If the precondition of G contains a contradiction (i.e. some goal is the negation of another goal), then this set of CHANGE PLANs are unintegrable, and the PCRM system notifies the CR manager and the programmers that the CHANGE PLANs must be reformulated to enable successful integration.

Our algorithm proceeds by resolving conflicts and redundancies between CHANGE TASKs in different CHANGE PLANs. The conflicts that may arise during plan integration are *unsafe causal links* ,*resource deficits* and *open preconditions*, which are resolved by either adding precedence constraints, adjusting resource amounts or adding a new CHANGE TASK. *Redundant CHANGE TASKs* are resolved by deleting the redundant CHANGE TASK. These conflicts and redundancies are formally defined as follows:

- A causal link <i, p, j> is unsafe if there is some k that deletes p which might occur between i and j. Such a k is called a clobberer.

- A resource deficit <C, r> occurs when there is a consumer C of some resource r such that GRS(C, r) < Precond(C, r), where Precond(C, r) is the amount of r required by C and is 1 if r is a software resource.

- An open precondition is a pair <p, j> where p is a precondition of j and there is no causal link of the form <i, p, j>.

- A CHANGE TASK i is redundant if:

1) For each causal link of the form <i, p, j> there is a node k that asserts p such that k is possibly before j and possibly after each clobberer C of the causal link.
2) The set of resources used by i are included in the set of resources used by k.
3) For each resource r produced by i, there is a way to reorder consumers and producers of r to avoid any resource deficits with respect to r.

An Example

The following example illustrates intuitive interpretations and possible resolutions of the conflicts and redundancies defined above. Suppose that a team of programmers is working on an e-mail system. In this system, e-mail messages are (1) received, (2) converted into internal object-oriented data, and (3) saved into files. The modules responsible for these operations are GetMail, Objectifier, and SaveObject. Regarding these operations, there are three ISSUEs in the current release of the system:

ISSUEa: There is a minor bug in saving objects.

ISSUEb: Saving many objects is too slow.

ISSUEc: POP3(Post Office Protocol) support is needed.

To solve these ISSUEs, three CHANGE PLANs are created. Each CHANGE PLAN has one CHANGE TASK in it:

Ta: Fix the bug in saving objects

Tb: Speed up saving objects

Tc: Add new e-mail protocol POP3

Table 1 shows the detail of these CHANGE TASKs.

Name	Ta - Fix the bug in saving objects
Goal	solved(ISSUEa, Rc)
PreCond	exists(ISSUEa, Rc)
Resources	SaveObject(Rc)
Effects	ADD solved(ISSUEa, Rc)
	UPDATE_MODULE(Rc, SaveObject)
Description	This task is to fix a minor bug in the current release of the system by modifying SaveObject in the current release.

Name	Tb - Speed up saving objects
Goal	solved(ISSUEb, Rn)
PreCond	exists(ISSUEb, Rc)
Resources	SaveObject(Rc), Objectifier(Rc), GetMail(Rc)
Effects	ADD solved(ISSUEa, Rn) UPDATE_MODULE(Rn, SaveObject) UPDATE_MODULE(Rn, Objectifier) UPDATE_MODULE(Rn, GetMail)
Description	This is to refine the algorithm to save objects in files.

Name	Tc - Add new e-mail protocol POP3
Goal	solved(ISSUEc, Rn)
PreCond	exists(ISSUEc, Rc) includes(Rn, Objectifier(Rc))
Resources	GetMail(Rc)
Effects	ADD solved(ISSUEc, Rn) UPDATE_MODULE(Rn, GetMail)
Description	This is to enhance the mail interface to support the post office protocol version 3.

Table 1: CHANGE TASK Ta, Tb, and Tc

In Table 1, Rc and Rn means the current release of the system and the next release, respectively. SaveObject(Rc) means a version of the module SaveObject used in the current release of the system. UPDATE_MODULE(R, M) replaces a version of the module M in the release R with a newly created version.

The following conflicts or redundancies, therefore, may arise in this example.

- Open precondition

Tc assumes there is a new release that includes the same version of Objectifier as the current release does. One of the possible solutions is:

* to add a new task Td that tentatively initializes a new release.

- Redundancy

If ISSUEb can cover ISSUEa, Ta is redundant to Tb. Possible solutions may include:

* to withdraw Ta
* to merge Ta and Tb

- Unsafe causal link

While Tc updates GetMail with an assumption that the version of Objectifier in the current release is also in the next release, Tb updates Objectifier in the next release. A causal link:

<Td, Tc, includes(Rn, Objectifier(Rc))>

is, therefore, unsafe with Tb. Possible solutions may include:

* to merge Tb and Tc
* to move Tb after Tc
  (i.e. to add the precedence constraints Tc < Tb)

- Resource deficit

Both Ta and Tb modify SaveObject. Some of possible solutions are:

* to move Ta after Tb or vice versa
* to increase the amount of the resource and the GRS values in Ta and Tb
  (i.e. to allow a parallel development)
* to merge Ta and Tb

Another resource deficit may occur among Tb and Tc regarding GetMail and can be handled in a similar way.

Based on possible interpretations of conflicts and redundancies, the PCRM system suggests a set of solutions to programmers. The final decision by the programmers may vary from a low level one like

ordering their tasks to a high level one like merging their tasks and changing their original strategy. Sometimes programmers may decide to defer solving conflicts or redundancies until execution time. Even in such a case, the integration process is useful in a sense that the programmers can be aware of the conflicts or redundancies in their tasks priori to executions.

THE CHANGE PLAN INTEGRATION ALGORITHM

Now, we formally define the plan integration algorithm used in the PCRM system. Figure 3 shows the change plan integration algorithm. The system defined actions are printed in non-bold italics. There are two execution modes in our algorithm. In the interactive mode, the algorithm produces one integrated plan through interaction with the CR manager and negotiation among programmers; in the batch mode, the algorithm produces possible integrated plans and lists of necessary negotiations to be performed. In this paper, we mainly describe how the algorithm works in the interactive mode.

STEP 1) *Do COLLECT* CHANGE PLANs into the Plan Work Space (PWS)
STEP 2) *Do INIT* with the CHANGE PLANs in PWS
STEP 3) Remove some plan Q in PWS that has conflicts
or redundant tasks.
If there is no such a Q, *Do DONE*.
STEP 4) If Q has open preconditions, repeat STEP-OP until
here is no open precondition.
go to STEP 3
STEP 5) If Q has redundant tasks, repeat STEP-RN until
there is no redundant task.
go to STEP 3
STEP 6) If Q has unsafe causal links, repeat STEP-UC until
there is no unsafe causal link.
go to STEP 3
STEP 7) If Q has resource deficits, repeat STEP-RD until
there is no resource deficit.
go to STEP 3

Conflict handler subroutines are defined as follows:
STEP-OP) (an open precondition : <P,j>)
Try NEW and *MOVE(i, j)* where i adds P
return
STEP-UC) (an unsafe causal link: <i,P,j> with a clobberer k)
Try MOVE(k, i) and MOVE(j, k)
return
STEP-RD) (a resouce deficit between i and j regarding r)
Try MOVE(i, j) , *MOVE(j,i)*, and *MODR(s,r)* where s is one of the suppliers of r
return
STEP-RN) (redundunt tasks {r1,r2,...,rn})
Try OUT(ri)
if temporary conflicts arises
Repeat STEP-UC until there is no unsafe causal link
Repeat STEP-OP until there is no open precondition
Repeat STEP-RD until there is no resource deficit
return

Figure 3: Change Plan Integration Algorithm

In the algorithm described in Figure 3, ***TRY*** has different semantics in two execution mode. In the batch mode, it means "Construct a new plan Q' by applying every action specified in the arguments to plan Q and add Q' to the plan work space". Neither interaction nor negotiation occurs. On the other hand, in the interaction mode, it put the actions into a list called a possible action list and invokes the interactive execution procedure we will describe later. ***Do*** is also performed interactively in the interactive mode, while it simply applies the action of its argument in the batch mode.

Table 2 shows the system defined actions in the PCRM system. Although some of them are not explicitly in the algorithm described above, they are in a possible action list as default actions in the interactive mode, so that programmers can choose one of the every possible actions the PCRM can execute.

Action	Description
INIT(CPs)	Initialize PWS with CHANGE PLANs (CPs)
OUT(i)	Withdraw a task i in PWS
MERGE(i,j)	Merge two tasks into one
MOVE(i,j)	Add a precedence constraint a task i < a task j
MODR(i, r)	Increase supply of a resource r in a task i
NEW	Add a newly created task to PWS
NOP	Defer resolution by the execution time
COLLECT	Collect CHANGE PLANs
DONE	Finish integration

Table 2: System Defined Actions in the PCRM system

ACTION EXECUTION THROUGH NEGOTIATION

In the interaction mode, an action to resolve a conflict or redundancy is not executed immediately but stored in a possible action list by the ***TRY*** operation. The execution process consists of several phases:

1. *Action prioritization.* The PCRM system tries to justify the actions in the action list using the knowledge base and sorts them according to the results of the justifications.

2. *Negotiation.* The system attempts to mediate between the potentially affected members and achieve a commitment from them about the resolution

3. *Execution.* The plan is modified by the selected action. Each execution is logged for backtracking.

1. Action Prioritization Phase

The PCRM system has customizable strategies to select appropriate actions. Each action has its own justification rules which consist of: the technical justification rules, the managerial justification rules

and the local justification rules. The PCRM system tries to prove the predicates in the rules which support the execution of an action. If some of the predicates are found to be true, it means that the corresponding action is very likely to be accepted. Figure 4 shows an example of a part of such rules for the action

```
(<- (can-move ?x ?y ?reason)
    (managerial-move ?x ?y ?reason))
(<- (can-move ?x ?y ?reason)
    (technical-move ?x ?y ?reason))
(<- (can-move ?x ?y ?reason)
    (other-move ?x ?y ?reason))
(<- (managerial-move ?x ?y ?reason)
    (more-important-task ?x ?y ?reason))
(<- (managerial-move ?x ?y ?reason)
  .....
(<- (more-important-task ?x ?y ?reason)
    (issues ?x ?ix) (from ?ix ?person-x)
    (issues ?x ?iy) (from ?iy ?person-y)
    (more-important ?person-x ?person-y))
```

MOVE. This example represents the fact that a CHANGE TASK Tx derived from a person X's report might precede a CHANGE TASK Ty derived from a person Y's report if the person X has a relation "more-important" with the person Y. This rule justifies MOVE(Tx,Ty) with a high priority. Primitive relations like "more-important" could be defined explicitly in the knowledge base or in the corresponding objects using the object link mechanism in Oval.

Figure 4: A Part of Justification Rules for "MOVE"

2. Negotiation Phase

After justifying the actions, the PCRM system selects an action of the highest priority. To achieve a commitment to execute the action, the system tries to establish conversation channel among the members selected by A) *role definitions* and B) *the negotiation table* described below.

A) Role definitions.

Members of a project can have the following roles.

- OWNER. <OWNER, anObject>
  An OWNER of a task, a resource, or an action is a person who is primarily responsible for the object.

- SUBSCRIBER. <SUBSCRIBER, anObject>
  A SUBSCRIBER of a task, a resource, or an action is a person who is interested in the object's state.

- DELEGATE.<DELEGATE, <a role definition>,anAction>
  A delegate of someone's role with an action is a person who perform the role when the action is to be executed.

The PCRM system has several default rules to define OWNER. For example, an OWNER of an object is assigned to a person whose PERSON object is in the ORIGINATOR or PERSON RESPONSIBLE field of the object. The CR manager has the OWNER roles of every ACTION objects.

On the other hand, SUBSCRIBER has more dynamic aspects. When an executed action is a binary operation (i.e. it has two tasks as its arguments), the PCRM system assigns each OWNER of the tasks to a SUBSCRIBER of the other after the execution. If the version and configuration management system could provide information on dependencies among resources, SUBSCRIBER roles could be defined to reflect the dependencies. For instance, if a resource A "depends on" a resource B, the PCRM system would assign the OWNER of the resource A to the SUBSCRIBER of the resource B.

Any role of a person can be delegated to anyone by the person or the PCRM operator. Example of how to use a DELEGATE role is discussed later.

B) The negotiation table

This table is used to select participants of a negotiation. Table 3 shows one of the default configurations of the negotiation table setting.

Each entry in the negotiation table corresponds to an action and shows who should be asked to get a commitment of the action execution and who should be notified. A value of a cell <action,person(s)> in the table specifies one of the following kinds of interaction or notification, regarding the event of execution of the action.

- ASK — The person(s) are asked before the event.
- NOTIFY-B — The person(s) are notified before the event.
- NOTIFY-A — The person(s) are notified after the event.
- NOTIFY — The person(s) are notified both before and after the event.
- TENTATIVE — Proceeds with a *tentative approval.* (described later)

Persons / Action	OWNER of the action	OWNERs of the tasks	SUBSCRIBERs of the tasks	OWNERs of the resources	SUBSCRIBERs of the resources
INIT(CPs)	ASK	NOTIFY-A			
OUT(i)	ASK	ASK	NOTIFY	NOTIFY-A	
MERGE(i,j)	ASK	ASK	NOTIFY-B	NOTIFY-A	
MOVE(i,j)	ASK	ASK	NOTIFY-A		
MODR(i, r)	ASK	ASK	NOTIFY	ASK	NOTIFY
NEW	ASK	NOTIFY-A			
NOP	ASK	ASK	NOTIFY-A	NOTIFY-A	NOTIFY-A
COLLECT	NOTIFY-A	NOTIFY-A			
DONE	ASK	NOTIFY-A		NOTIFY-A	

Table 3: An Example of the Negotiation Table

- no value No interaction required.

If the PCRM system finds that multiple persons should be asked for a commitment, the system ask them to negotiate each other to arrive at a consensus about the commitment. In Table 3, for example, the entry for MERGE(i,j) shows: 1) OWNER of the action (i.e. the CR manager as a default) and OWNERs of the task i and task j should be asked by the system about the MERGE action; 2) SUBSCRIBERs of both tasks are notified before the negotiation; 3) if the negotiation is done successfully, OWNERs of the resources used in both tasks are notified after the MERGE action.

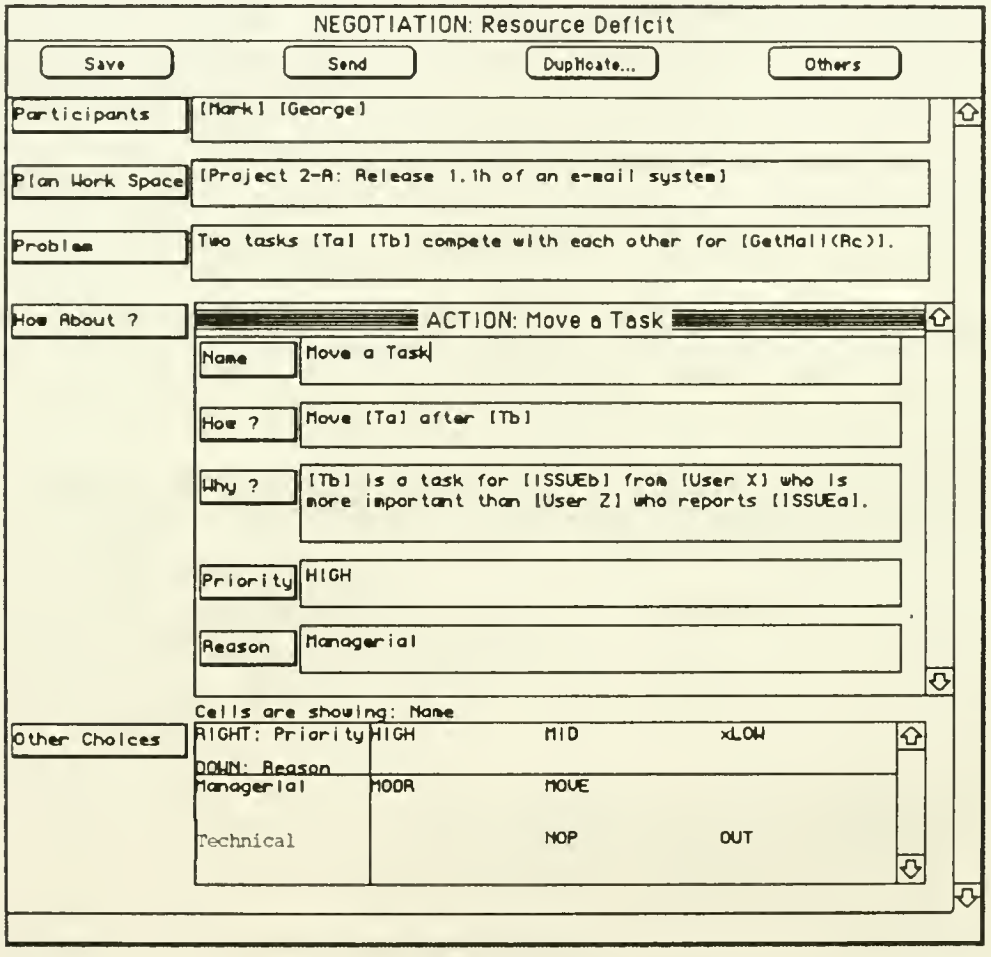

Figure 5: An Example of a NEGOTIATION object sent to programmers.

According to the table, the PCRM system creates a NEGOTIATION object and sends it to relevant programmers to initiate a communication channel. Figure 5 shows a NEGOTIATION object that is sent to programmers at a negotiation in the example described earlier in Table 1.

This object contains information about the highest priority action "MOVE" to solve a resource deficit. It also contains links to other actions in the possible action list.

3. Execution Phase

The PCRM system repeats negotiations until one of them succeeds. In case that every attempts failed, the PCRM manager can proceed by either repeating the negotiation phase again, using a *tentative approval*, or executing one of the action with the manager's authority. Every execution is recorded for the backtracking.

Tentative Approval and Backtracking

In case the PCRM system cannot get an immediate answer from members who are busy or away from their work, the integration phase could proceed with a tentative approval. If (1) the CR manager tells the system to continue the integration without waiting the answer from the members, or (2) a cell <action, person(s)> in the negotiation table has a value TENTATIVE instead of ASK, the system automatically proceeds with a tentative approval of the action. A tentative approval can be changed to real one during the integration. The system checks if there is any tentative approval left in the plan when it executes DONE action at the end of the integration. A simple backtracking mechanism is supported and the system could restore any previous state of the integration.

REPEATING INTEGRATION

When programmers are performing some of CHANGE TASKs in an integrated plan after an integration, they may need to modify the plan due to the later changes in the situation. In such a case, they can set the status of the CHANGE TASKs in the plan to either *active, dead or inactive* , which means being executed, being withdrawn, or waiting for execution respectively. This integrated plan with the status information can be re-integrated with other new CHANGE PLANs at the next integration. The status information plays an important role in both the action prioritization phase and the negotiation phase. The standard justification rules include rules such as *"Active tasks should not be modified as less as possible.", "Dead tasks should be withdrew first.", and "Inactive tasks have a higher priority than new tasks because inactive tasks have already been approved"*. The problem to adapt a plan to a changing environment is known as replanning [26]. The strategy described above is a way of replanning in our architecture with human assistance.

THE KNOWLEDGE BASE

The PCRM knowledge base consists of three major parts: 1) conflict resolution strategies which includes the main algorithm; 2) the coordination strategies which includes the actions, the justification rules, the roles, and the negotiation table; 3) the project knowledge such as relationship among users of a target software or the structure of a development team. In the current implementation, knowledge is represented in one of two ways:

- Any Oval object and its relationship can be used to represent knowledge. A knowledge base object contains links to Oval object's types. The PCRM system periodically checks changes in objects of the

types and incorporate their information into the knowledge. Each object knows what parts of its information or fields should be exported as knowledge. This mechanism enables user to combine the PCRM system with the existing Oval applications [14] such as employee databases, decision making systems or schedule management.

- A knowledge object in a knowledge base object contains knowledge in a Prolog notation. The conflict resolution strategies and the coordination strategies are represented in this way.

CUSTOMIZATION OF THE CHANGE REQUEST MANAGEMENT PROCESS

Different project has different polices on how to deal with software changes. The PCRM system can be tailored in the following ways.

Customizing Change Request Information

Because every data in the PCRM system including CHANGE PLAN, CHANGE TASK, and ISSUE is defined as a semi-structured object in Oval which is a "radically tailorable[16]" system, it is quite easy to customize: (1) their definitions by adding or deleting user-defined fields, (2) relationships among them by defining links, and (3) views of them by selecting appropriate display formats.

Customizing the Life Cycle of Changes

In Figure 6, the *PCRM life cycle editor* , that is a part of the user interface agent, displays a life cycle view of a CHANGE TASK. This state transition is automatically derived from the definition of the actions in the PCRM system. The life cycle editor helps the CR manager understand the life cycle and customize the state transition by (1) defining a new action and (2) modifying the existing actions.

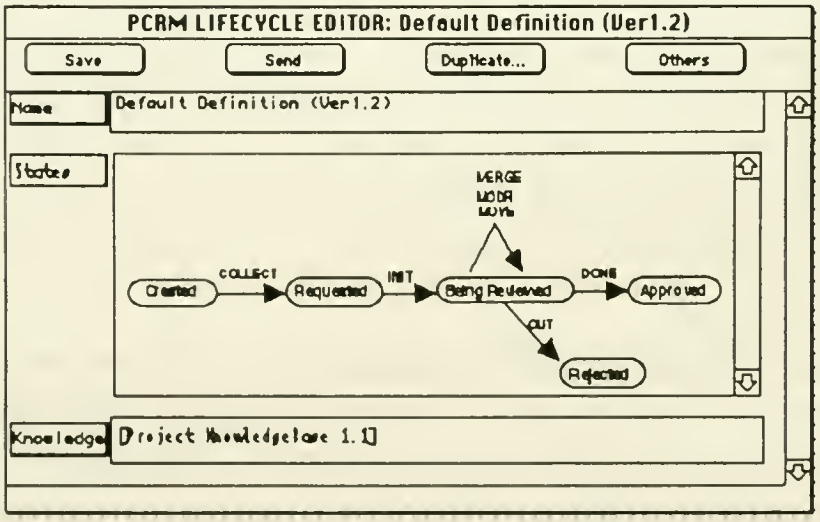

Figure 6: The PCRM Life Cycle Editor.

It shows a state diagram where each node corresponds to a state of a CHANGE TASK and each arrow represents an action in the PCRM system.

Any pre- and post-integration jobs can be defined as a new action that is executed by an Oval agent. Suppose every CHANGE PLAN in a project should be analyzed by a project manager to estimate the cost of its change, prior to the integration. This can be modeled by defining a new action and assigning the manager to the OWNER of the action. The life cycle editor creates an action object and also generates an Oval agent which executes the action whenever the status of any CHANGE PLAN becomes REQUESTED. When the action is to be executed, the system creates a communication channel to the manager so that he/she can do his/her job with the CHANGE PLANs.

The system defined actions that are executed by the plan integrator can be also customized by creating a subtype of them. Although some characteristics of them cannot be changed, user can customize, for example, how to change the status of a CHANGE TASK when a negotiation for an action fails.

Customizing Negotiations

Role definitions and the negotiation table define how negotiations should be done. The PCRM system provides several standard configurations which can be customized by changing role assignment or changing values of the cells in the negotiation table. It is also possible to model a new role who performs a special job in negotiations such as technical reviewer or configuration manager. For example, suppose a member in a project has a role

<DELEGATE, <OWNER, aCHANGE_TASK> MODR>

for every CHANGE TASK. This means that the member is treated as OWNERs of every CHANGE TASK when the MODR action (i.e. the action which increases the amount of a resource) is being executed. If the entry of the MODR action in the negotiation table has the ASK attribute for OWNERs of the tasks, the system asks the member to give a permission every time it attempts to execute the MODR action over the CHANGE TASKs. Because the execution of MODR action with a software module as a resource may mean creating a version branch of the module, this negotiation ensures the member who is a configuration manager can have a chance to decide whether a version branch is required or not.

Customizing Action Prioritization

The justification rules of the actions are used to prioritize actions and ,therefore, determine the strategies on how to deal with conflicts and redundancies. The rules have predicates over relationships among Oval objects that represents the project knowledge. The primary level modifications to the action prioritization can be done by modifying the fact information in these Oval objects. The another level modifications can be done by customizing the rules which determine the interpretation of the fact information. The PCRM system has rule libraries to help the CR manager to construct the justification rules.

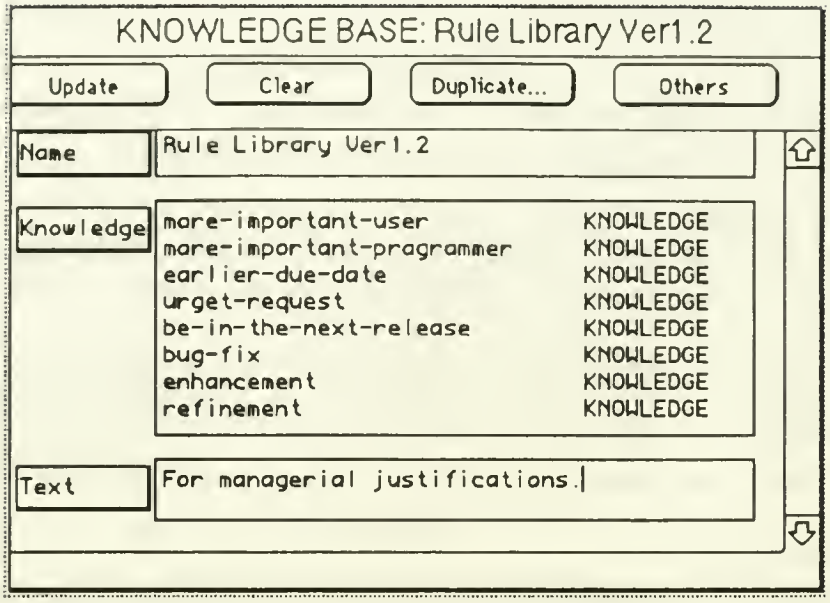

Figure 7: A Knowledge Base object that contains a library for managerial justification rules

Figure 7 shows such a library for managerial justification rules. Using these primitive predicates, the CR manager can define a new justification rule or customize the standard justification rules to meet project's particular requirements.

DISCUSSION

One of the most important factors in describing a CHANGE PLAN is the abstraction level of the CHANGE TASKs. Suppose there is a CHANGE PLAN named "Speed-up Tuning" to solve an ISSUE named "The system X is slow" and it has a very high level CHANGE TASK which uses every software module of the system X. This CHANGE PLAN would cause resource deficits with every other CHANGE PLAN at the integration time. In such a case, the high level task should be decomposed into reasonably lower level ones before the integration. Our architecture currently doesn't have hierarchical planning method which could handle the different abstraction levels of CHANGE TASKs. To decide the optimal abstract level of CHANGE TASKs is closely related to the abstract level of ISSUEs, and is essentially the granularity problem that involves the size of the process elements in software process program [5]. Although we do not explicitly model the creative portion of constructing CHANGE PLANs, we believe that the abstract level of CHANGE TASKs gradually converges into a certain range as the project matures [11] and the target software becomes stable.

RELATED WORK

K.Narayanaswamy, et al, [21] describe an architecture where proposed changes are gradually approved by affected programmers. The causal relationships between proposed changes are maintained so that negotiations can be supported. Although their negotiation protocol could be modeled in our customizable representation of a change management process, their architecture has a database transaction

concept which is currently out of the scope of our work. Another difference is that our system can model the project's policies on change request management.

Softbench [4] features an event notification mechanism that notifies a change event when the change has been made. It is the opposite approach of notification compared to ours.

Kaiser [12] has extended the traditional database transaction models to support the notion of long-term transactions. Their concepts for cooperative development: *activity interaction* and *programmer interaction* could provide a very good basis to combine our framework with execution environments because these concepts could fit well to our support of replanning with the task status information.

There are several systems that support the life cycle of change requests. Lifespan [25] provides notification mechanism based on the life cycle. CCC [7] provides a life cycle support of change requests by separating the life cycle into four phases. PCMS [19] concentrates on providing flexibility for the life cycle of software items and change requests. In these system, however, the coordination among the programmers is limited by the capability of their configuration management systems which are based on the check-in check-out model [9]. Madhavji [15] defines a universal model of changes in software development environments. The model can represent changes in organizations and project policies. Conflict and redundancy management could be built on their model.

Huff, et al, [10] apply the planning paradigm to software development, featuring a truth maintenance system. Croff, et al, [6] use negotiation among agents including human to handle the exception happens during plan execution. Unlike our system, these systems focus on intelligent plan execution process. Martial [17] describes a conversation model for resolving conflicts using temporal relationship among plans of office activities. Although the model has a similar negotiation mechanism, it doesn't use any domain specific knowledge and its conflict resolution strategies are not customizable.

Perry, et al, [23] discuss a general model of software development. It consists of three components: policies, mechanisms and structures. One of our goals is to support what they call *enforced cooperation*. Systems related to policy enforcement such as ISTAR [8], Darwin [18], and [19] can provide us various ideas how to represent project polices.

CONCLUSION

In this paper, we have described a change request management system which supports integration of change plans into a consistent plan. The integration algorithm and strategies to resolve conflicts and redundancies were described in detail. The current implementation is based on Oval with several enhancements written in CommonLisp and running on networked Apple Macintosh.

In the future we expect to work on the following aspects of the PCRM system:

- Experimentation

We would like to test our system in practical situations to study effective negotiation protocols, as well as to explore representations of project policies on change request management.

- Integration with version and configuration management

We plan to extend our architecture by incorporating a version and configuration management system or software database.

- Reasoning about time

Temporal reasoning and scheduling capability should be added to our system to support project management of software development.

- Recording the reasons

Information about a negotiation process is very useful at an execution time as well as after the execution, because it could explain how and why the decision was made in the negotiation process. With a combination of group decision making support such as [14], the PCRM system could provide a mechanism to record the reasons of changes.

REFERENCES

[1] Adams, E., Honda, M., and Miller, T. Object Management in a Case Environment. in *Proc. of 11th International Conference on Software Engineering*, IEEE, 1989.

[2] Ayer, S. J. and Patrinostro, F. *Software Configuration Management: Identification, Accounting, Control, & Management*, McGraw-Hill, Inc., 1992.

[3] Berliner, B. CVS II: Parallelizing Software Development, in *Proc. of USENIX'90*, 1990

[4] Cagan, M. The HP Softbench Environment: An Architecture for a New Generation of Software Tools. in *Hewlett-Packard Journal*, 1990.

[5] Curtis, B., Kellener, M. I., Over, J. *Process Modeling, in Communications of ACM*, Sep. 1992.

[6] Croft, W. B. and Lefkowitz, L. S. Knowledge-Based Support of Cooperative Activities, in *Proc. of 21st International Hawaii Conference on System Science*, 1988

[7] Dart S. Concepts in Configuration Management Systems, in *Proc. of International Workshop on Software Version and Configuration Control,* ACM, 1991.

[8] Dowson, M. Integrated Project Support with IStar, in *IEEE Software*, Nov. 1987

[9] Feiler, P. H. Configuration Management Models in Commercial Environments., *CMU Software Engineering Institute Technical Report*, CMU/SEI-91-TR7, Pittsburgh, PA, March 1991.

[10] Huff, K. E. and Lesser, V. R. A Plan-based Intelligent Assistant That Supports the Software Development Process, in *Proc. of the 3rd Software Engineering Symposium on Practical Software Development Environments*, ACM, 1989.

[11] Humphrey, W. *Managing the Software Process*, SEI Series in Software Engineering, Addison-Wesley Publishing Company, 1989.

[12] Kaiser, G. A Flexible Transaction Model for Software Engineering. in *Proc. of 6th International Conference on Data Engineering*, IEEE, 1990.

[13] Lee, J. Sibyl: A tool for managing group design. in *Proc. of CSCW'90*, ACM, 1990

[14] Lee, G. *Oval Application Catalogue*, MIT Center for Coordination Science Technical Report, 1 Amherst St. Cambridge, MA., Sep. 1992

[15] Madhavji, N. H. The Prism Model of Changes, in *Proc. of 13th International Conference on Software Engineering*, IEEE, 1991.

[16] Malone, T., Lai, K., and Fry, C. Experiments with Oval: A radically Tailorable Tool for Cooperative Work, in *Proc. of CSCW '92*, ACM, 1992.

[17] Martial, F. A Conversation Model for Resolving Conflicts among Distributed Office Activities., in *Proc. of Conference on Office Information Systems*, ACM, 1990.

[18] Minsky, N. H. Law-Governed Software Process, in *Proc. of International Software Process Workshop*, IEEE, 1990

[19] Moffet, J. D. and Sloman, M. S. The Representation of Policies as System Objects, in *Proc. of Conference on Office Information Systems*, ACM, 1991.

[20] Montes, J. and Haque, T. A. Configuration Management System and More!, in *Proc. of International Workshop on Software Version and Configuration Control*, ACM, 1988.

[21] Narayanaswamy, K. and Goldman, N. "Lazy" Consistency: A Basis for Cooperative Software Development, in *Proc. of CSCW '92*, ACM, 1992.

[22] Harrison, W., Ossher, H., Seeney, P. Coordinating Concurrent Development. in *Proc. of CSCW '90*, ACM, 1990.

[23] Perry, D. E., Kaiser, G. Models of Software Development Environments, in *Proc. of 10th International Conference on Software Engineering*, IEEE, 1988.

[24] Rosenblitt, D. *Supporting collaborative planning: The Plan Integration Problem*, Ph.D. thesis, Department of Electrical Engineering and Computer Science, M.I.T., 1991.

[25] Whitgift, D. *Method and Tools for Software Configuration Management*, Wiley series in software engineering practice, John Wiley & Sons, Inc., 1991.

[26] Willkins, D. E. *Practical Planning: Extending the Classical AI Planning Paradigm*, Morgan Kaufmann, 1988.

[27] Winograd, T. A Language/Action Perspective on the Design of Cooperative Work. in *Computer Supported Cooperative Work: A Book of Readings.* I. Greif, Ed. Morgan Kaufmann, San Mateo, CA, 1988.